Building Bedrock: Verifying a Program Verifier

Gregory Malecha (gmalecha@cs.harvard.edu)
Adam Chlipala, Thomas Braibant, Patrick Hulin, Edward Yang (MIT)

Harvard SEAS

Coq Workshop – Aug 13, '12

BEDROCK

Automated Verification Framework Coq

BEDROCK

Automated Verification **Framework** Coq

User extensible.

Bedrock

Automated Verification Framework Coq

How we built it

What we learned about reflective automation.

```
Definition hints : TacPackage.
  prepare (bst_fwd, nil_fwd, cons_fwd) (bst_bwd, nil_bwd, cons_bwd).
Defined.

Definition bstM := bmodule "bst" {{
  bfunction "lookup"("s", "k", "tmp") [lookupS]
    "s" ←∗ "s";;
    [Ex s, Ex t,
      PRE[V] bst' s t (V "s") ∗ mallocHeap
      POST[R] ⌈ (V "k" ∈ s) \is R ⌉ ∗ bst' s t (V "s") ∗ mallocHeap]
    While ("s" ≠ 0) {
      "tmp" ← "s" + 4;;
      "tmp" ←∗ "tmp";;
      If ("k" = "tmp") {
        (* Key matches! *)
        Return 1
      } else {
        If ("k" < "tmp") {
          (* Searching for a lower key *)
          "s" ←∗ "s"
        } else {
          (* Searching for a higher key *)
          "s" ← "s" + 8;;
          "s" ←∗ "s"
        }
      }
    };;
    Return 0
  end }}.

Theorem bstMOk : moduleOk bstM.
Proof. vcgen; abstract (sep hints; auto). Qed.
```

```
Definition hints : TacPackage.
  prepare (bst_fwd, nil_fwd, cons_fwd) (bst_bwd, nil_bwd, cons_bwd).
Defined.

Definition bstM := bmodule "bst" {{
  bfunction "lookup"("s", "k", "tmp") [lookupS]
    "s" ←∗ "s";;
    [Ex s, Ex t,
      PRE[V] bst' s t (V "s") ∗ mallocHeap
      POST[R] ⌈ (V "k" ∈ s) \is R ⌉ ∗ bst' s t (V "s") ∗ mallocHeap]
    While ("s" ≠ 0) {
      "tmp" ← "s" + 4;;
      "tmp" ←∗ "tmp";;
      If ("k" = "tmp") {
        (* Key matches! *)
        Return 1
      } else {
        If ("k" < "tmp") {
          (* Searching for a lower key *)
          "s" ←∗ "s"
        } else {
          (* Searching for a higher key *)
          "s" ← "s" + 8;;
          "s" ←∗ "s"
        }
      }
    };;
    Return 0
  end }}.

Theorem bstMOk : moduleOk bstM.
Proof. vcgen; abstract (sep hints; auto). Qed.
```

Imperative code

Hoare logic-like specifications

Separation logic

```
Definition hints : TacPackage.
  prepare (bst_fwd, nil_fwd, cons_fwd) (bst_bwd, nil_bwd, cons_bwd).
Defined.

Definition bstM := bmodule "bst" {{
  bfunction "lookup"("s", "k", "tmp") [lookupS]
    "s" ←∗ "s";;
    [Ex s, Ex t,
      PRE[V] bst' s t (V "s") ∗ mallocHeap
      POST[R] ⌈ (V "k" ∈ s) \is R ⌉ ∗ bst' s t (V "s") ∗ mallocHeap]
    While ("s" ≠ 0) {
      "tmp" ← "s" + 4;;
      "tmp" ←∗ "tmp";;
      If ("k" = "tmp") {
        (* Key matches! *)
        Return 1
      } else {
        If ("k" < "tmp") {
          (* Searching for a lower key *)
          "s" ←∗ "s"
        } else {
          (* Searching for a higher key *)
          "s" ← "s" + 8;;
          "s" ←∗ "s"
        }
      }
    }
  end }}

Theorem bstMOk : moduleOk bstM.
Proof. vcgen; abstract (sep hints; auto). Qed.
```

Verification

```
Definition hints : TacPackage.
  prepare (bst_fwd, nil_fwd, cons_fwd) (bst_bwd, nil_bwd, cons_bwd).
Defined.

Definition bstM :=
  bfunction "lookup"("s", "k", "tmp") [lookupS]
    "s" ←∗ "s";;
    [Ex s, Ex t,
      PRE[V] bst' s t (V "s") ∗ mallocHeap
      POST[R] ⌈ (V "k" ∈ s) \is R ⌉ ∗ bst' s t (V "s") ∗ mallocHeap]
    While ("s" ≠ 0) {
      "tmp" ← "s" + 4;;
      "tmp" ←∗ "tmp";;
      If ("k" = "tmp") {
        (* Key matches! *)
        Return 1
      } else {
        If ("k" < "tmp") {
          (* Searching for a lower key *)
          "s" ←∗ "s"
        } else {
          (* Searching for a higher key *)
          "s" ← "s" + 8;;
          "s" ←∗ "s"
        }
      }
    };;
    Return 0
  end }}.

Theorem bstMOk : moduleOk bstM.
Proof. vcgen; abstract (sep hints; auto). Qed.
```

User-extensible hints

Automation

Proofs By Tactics

$$\boxed{\langle \text{Goal} \rangle} \xrightarrow{\text{tactics}} \boxed{\langle \text{Goal} \rangle}$$

- ✗ Hard work!
- ✗ Big proofs!

Proofs By Ltac

$$\boxed{\langle\text{Goal}\rangle} \xrightarrow{\text{ltac}} \boxed{\langle\text{Goal}\rangle}$$

- ✓ **Automatic**
- ✗ Big proofs!

Proofs By Reflection

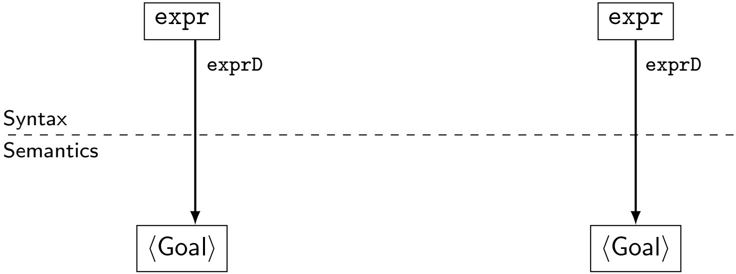
`expr` `expr`

Syntax

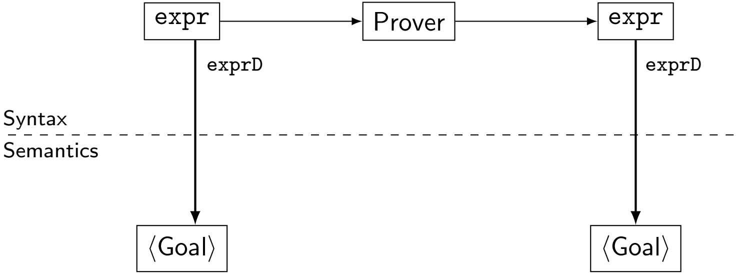
Semantics

$\langle$Goal$\rangle$ $\langle$Goal$\rangle$

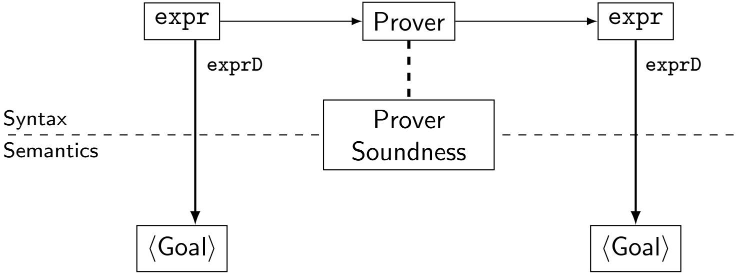
Proofs By Reflection



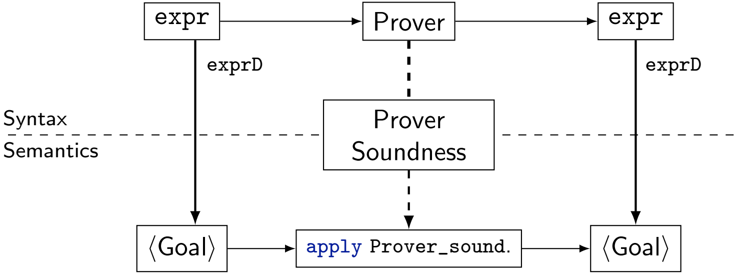
Proofs By Reflection



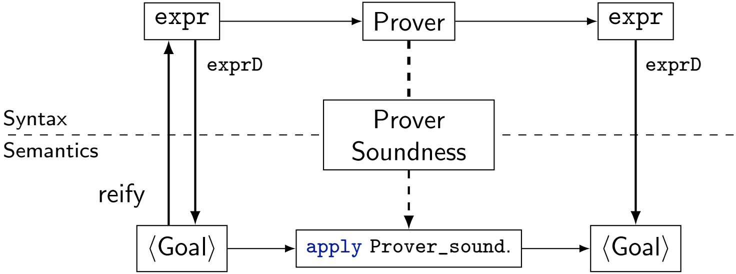
Proofs By Reflection



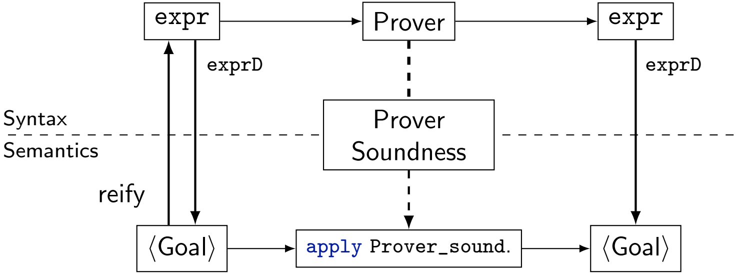
Proofs By Reflection



Proofs By Reflection



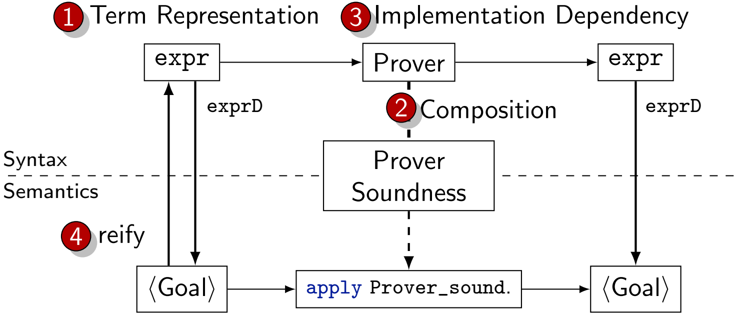
Proofs By Reflection



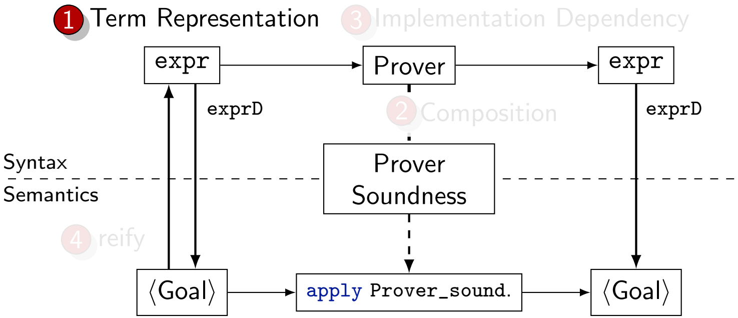
✓ **Automatic**

✓ **Small proofs**

Outline


1 Term Representation
3 Implementation Dependency
expr
Prover
expr
exprD
2 Composition
exprD
Syntax
Semantics
Prover
Soundness
4 reify
⟨Goal⟩
apply Prover_sound.
⟨Goal⟩

Outline: Term Representation



Representing Terms

```
Variable ts : list Type.
Inductive expr : Type :=
| Const : ∀ t, tvarD ts t → expr
| Var : nat → expr
| UVar : nat → expr
| Func : nat → list expr → expr
| Equal : tvar → expr → expr → expr.
```

Representing Terms

Type environment

Type denotation function

```
Variable ts : list Type.
Inductive expr : Type :=
| Const : ∀ t, tvarD ts t → expr
| Var : nat → expr
| UVar : nat → expr
| Func : nat → list expr → expr
| Equal : tvar → expr → expr → expr.
```

Representing Terms

```
Variable ts : list Type.
Inductive expr : Type :=
| Const : ∀ t, tvarD ts t → expr
| Var : nat → expr
| UVar : nat → expr
| Func : nat → list expr → expr
| Equal : tvar → expr → expr → expr.
```

```
Variable fs : functions ts.
Variable menv venv : env ts.
Fixpoint exprD (e : expr) (t : tvar) :
     option (tvarD ts t) :=
  match e with
    | Const t' c ⇒
      cast_or_fail t t' c
    | Var x ⇒ lookupAs venv t x
    | UVar x ⇒ lookupAs menv t x
    | Func f xs ⇒
      match nth_error fs f with
        | None ⇒ None
        | Some f ⇒
          cast_or_fail t (Range f)
            (applyD exprD _ xs _ (Impl f))
      end
    | Equal t l r ⇒ ...
  end.
```

Representing Terms

```
Variable ts : list Type.
Inductive expr : Type :=
| Const : ∀ t, tvarD ts t → expr
| Var : nat → expr
| UVar : nat → expr
| Func : nat → list expr → expr
| Equal : tvar → expr → expr → expr.
```

```
Variable fs : functions ts.
Variable menv venv : env ts.
Fixpoint exprD (e : expr) (t : tvar) :
    option (tvarD ts t) :=
  match e with
    | Const t' c ⇒
      cast_or_fail t t' c
    | Var x ⇒ lookupAs venv t x
    | UVar x ⇒ lookupAs menv t x
    | Func f xs ⇒
      match nth_error fs f with
        | None ⇒ None
        | Some f ⇒
          cast_or_fail t (Range f)
            (applyD exprD _ xs _ (Impl f))
      end
    | Equal t l r ⇒ ...
  end.
```

Partial function

Representing Terms

```
Variable ts : list Type.
Inductive expr : Type :=
| Const : ∀ t, tvarD ts t → expr
| Var : nat → expr
| UVar : nat → expr
| Func : nat → list expr → expr
| Equal : tvar → expr → expr → expr.
```

```
Variable fs : functions ts.
Variable menv venv : env ts.
Fixpoint exprD (e : expr) (t : tvar) :
    option (tvarD ts t) :=
  match e with
    | Const t' c ⇒
      cast_or_fail t t' c
    | Var x ⇒ lookupAs venv t x
    | UVar x ⇒ lookupAs menv t x
    | Func f xs ⇒
      match nth_error fs f with
        | None ⇒ None
        | Some f ⇒
          cast_or_fail t (Range f)
            (applyD exprD _ xs _ (Impl f))
      end
    | Equal t l r ⇒ ...
  end.
```

Dependent types ⇒ many choices...

Other Choices?

	Minimal
Total	✗
Simple	✓

Other Choices?

	Minimal	Full
Total	✗	
Simple	✓	

Full Dependency

```
Variable ts : list Type.
Variables uenv venv : list tvar.
Variable fs : functions.
Inductive expr : tvar → Type :=
| Const : ∀ t, tvarD ts t → expr t
| UVar : ∀ t, Mem t uenv → expr t
| ...

Fixpoint exprD t (e:expr t)
  : tvarD ts t := ...
```

Other Choices?

	Minimal	Full
Total	✗	✓
Simple	✓	✗

Full Dependency

More dependency

```
Variable ts : list Type.
Variables uenv venv : list tvar.
Variable fs : functions.
Inductive expr : tvar → Type :=
| Const : ∀ t, tvarD ts t → expr t
| UVar : ∀ t, Mem t uenv → expr t
| ...
```

All terms are well-typed!

```
Fixpoint exprD t (e:expr t)
  : tvarD ts t := ...
```

Complex implementation

Other Choices?

	Minimal	Full
Total	✗	✓
Simple	✓	✗
Fast	✓	✗

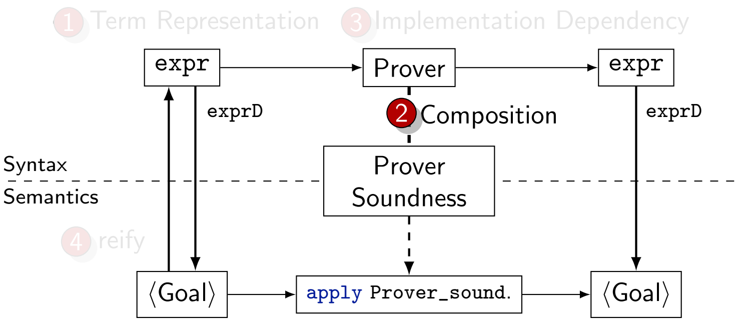
- More reduction for casts

Full Dependency

```
Variable ts : list Type.
Variables uenv venv : list tvar.
Variable fs : functions.
Inductive expr : tvar → Type :=
| Const : ∀ t, tvarD ts t → expr t
| UVar : ∀ t, Mem t uenv → expr t
| ...

Fixpoint exprD t (e:expr t)
  : tvarD ts t := ...
```

~20% speedup

Outline: Composition



Proving

Constant Fold +

```
Variable ts : list Type.
Fixpoint cfp (e : expr ts) : expr ts :=
  match e with
    | Const _ _ | Var _ | UVar _ ⇒ e
    | Func f args ⇒
      match f , map cfp args with
        | 0 , [Const 0 l, Const 0 r] ⇒
          Const 0 (l + r)
        | _ , args ⇒ Func f args
      end
  end.
```

Proving

Constant Fold +

```
Variable ts : list Type.
Fixpoint cfp (e : expr ts) : expr ts :=
  match e with
    | Const _ _ | Var _ | UVar _ ⇒ e
    | Func f args ⇒
      match f , map cfp args with
        | 0 , [Const 0 l, Const 0 r] ⇒
          Const 0 (l + r)
        | _ , args ⇒ Func f args
      en
  end.
```

Bad Type! tvarD ts 0

Proving

Consta

Easy fix...

```
Let ts := nat :: nil.
Fixpoint cfp (e : expr ts) : expr ts :=
  match e with
    | Const _ _ | Var _ | UVar _ ⇒ e
    | Func f args ⇒
      match f , map cfp args with
        | 0 , [Const 0 l, Const 0 r] ⇒
          Const 0 (l + r)
        | _ , args ⇒ Func f args
      end
  end.
```

Proving

Consta[...] Easy fix...

```
Let ts := nat :: nil.
Fixpoint cfp (e : expr ts) : expr ts :=
  match e with
    | Const _ _ | Var _ | UVar _ ⇒ e
    | Func f args ⇒
      match f , map cfp args with
        | 0 , [Const 0 l, Const 0 r] ⇒
          Const 0 (l + r)
        | _ , args ⇒ Func f args
      end
  end.
```

Constant Fold <

```
Let ts := nat :: bool :: nil.
Fixpoint cflt (e : expr ts) : expr ts :=
  match e with
    | Const _ _ | Var _ | UVar _ ⇒ e
    | Func f args ⇒
      match f , map cflt args with
        | 1 , [Const 0 l, Const 0 r] ⇒
          Const 1 (ltb l r)
        | _ , args ⇒ Func f args
      end
  end.
```

...but it doesn't compose any more!

Proving

Consta

Easy fix...

```
Let ts := nat :: nil.
Fixpoint cfp (e : expr ts) : expr ts :=
  match e with
    | Const _ _ | Var _ | UVar _ ⇒ e
    | Func f args ⇒
      match f , map cfp args with
        | 0 , [Const 0 l, Const 0 r] ⇒
          Const 0 (l + r)
        | _ , args ⇒ Func f args
      end
  end.
```

Constant Fold <

```
Let ts := nat :: bool :: nil.
Fixpoint cflt (e : expr ts) : expr ts :=
  match e with
    | Const _ _ | Var _ | UVar _ ⇒ e
    | Func f args ⇒
      match f , map cflt args with
        | 1 , [Const 0 l, Const 0 r] ⇒
          Const 1 (ltb l r)
        | _ , args ⇒ Func f args
      end
  end.
```

...but it doesn't compose any more!

Composition

```
Definition compose T (f g : T → T) : T → T := fun x ⇒ f (g x).
```

Achieving Composition

- Need to state "all environments with nat at 0"

Propositional

```
Variable ts : list Type.
Hypothesis natAt0 : tvarD 0 ts = nat.
Fixpoint cfp (e : expr ts) : expr ts :=
  match e with
  | Const _ _ | Var _ | UVar _ ⇒ e
  | Func f args ⇒
    match f , map red args with
    | 0 , [Const 0 l, Const 0 r] ⇒
      match natAt0 in _ = t
        return t → t → expr ts with
      | refl_equal ⇒ fun l r ⇒
        Const 0 (match sym_eq natAt0
                   in _ = t return t with
                 | refl_equal ⇒ (l + r)
                 end)
      end l r
    | _ , args ⇒ Func f args
    end
  end.
```

Achieving Composition

- Need to state "all e vith nat at 0"

Update ts' to satisfy the requirement

Update

```
Variable ts' : list Type
Let ts := repr (Some nat :: nil) ts'.
Fixpoint cfp (e : expr ts) : expr ts :=
  match e with
  | Const _ _ | Var _ | UVar _ => e
  | Func f args =>
    match f , map red args with
    | 0 , [Const 0 l, Const 0 r] =>
      match natAt0 in _ = t
        return t -> t -> expr ts with
      | refl_equal => fun l r =>
        Const 0 (match sym_eq natAt0
                      in _ = t return t with
                    | refl_equal => (l + r)
                    end)
      end l r
    | _ , args => Func f args
    end
  end.
```

Achieving Composition

- Need to state "all environments with nat at 0"

Update

```
Variable ts' : list Type.
Let ts := repr (Some nat :: nil) ts'.
Fixpoint cfp (e : expr ts) : expr ts :=
  match e with
  | Const _ _ | Var _ | UVar _ ⇒ e
  | Func f args ⇒
    match f , map red args with
    | 0 , [Const 0 l, Const 0 r] ⇒
      match natAt0 in _ = t
        return t → t → expr ts with
      | refl_equal ⇒ fun l r ⇒
        Const 0 (match sym_eq natAt0
                   in _ = t return t with
                 | refl_equal ⇒ (l + r)
                 end)
      end l r
    | _ , args ⇒ Func f args
    end
  end.
```

Environment Constraints

```
Variable T : Type.
Variable default : T.

Fixpoint repr (rep : list (option T)) :
    list T → list T :=
  match rep with
    | nil ⇒ fun x ⇒ x
    | None :: rep ⇒ fun x ⇒
      hd default x :: repr rep (tl x)
    | Some v :: rep ⇒ fun x ⇒
      v :: repr rep (tl x)
  end.
```

Achieving Composition

- Need to state "all environments with nat at 0"

Update

```
Variable ts' : list Type.
Let ts := repr (Some nat :: nil) ts'.
Fixpoint cfp (e : expr ts) : expr ts :=
  match e with
  | Const _ _ | Var _ | UVar _ ⇒ e
  | Func f args ⇒
    match f , map red args with
    | 0 , [Const 0 l, Const 0 r] ⇒
      match natAt0 in _ = t
        return t → t → expr ts with
      | refl_equal ⇒ fun l r ⇒
        Const 0 (match sym_eq natAt0
                   in _ = t return t with
                 | refl_equal ⇒ (l + r)
                 end)
      end l r
    | _ , args ⇒ Func f args
    end
  end.
```

Environment Constraints

```
Variable T : Type.
Variable default : T.

Fixpoint repr (rep : list (option T)) :
    list T → list T :=
  match rep with
    | nil ⇒ fun x ⇒ x
    | None :: rep ⇒ fun x ⇒
      hd default x :: repr rep (tl x)
    | Some v :: rep ⇒ fun x ⇒
      v :: repr rep (tl x)
  end.
```

Properties

$$\text{repr}\ l\ (\text{repr}\ r\ ls) = \text{repr}\ r\ (\text{repr}\ l\ ls)$$
$$\text{repr}\ l\ (\text{repr}\ l\ ls) = \text{repr}\ l\ ls$$

Achieving Composition

- Need to state "all environments with nat at 0"

Update

```
Variable ts' : list Type.
Let ts := repr (Some nat :: nil) ts'.
Fixpoint cfp (e : expr ts) : expr ts :=
  match e with
  | Const _ _ | Var _ | UVar _ ⇒ e
  | Func f args ⇒
    match f , map red args with
    | 0 , [Const 0 l, Const 0 r] ⇒
      match natAt0 in _ = t
        return t → t → expr ts with
      | refl_equal ⇒ fun l r ⇒
        Const 0 (match sym_eq natAt0
                  in _ = t return t with
                 | refl_equal ⇒ (l + r)
                 end)
      end l r
    | _ , args ⇒ Func f args
    end
  end.
```

Environment Constraints

```
Variable T : Type.
Variable default : T.

Fixpoint re                    ) :
    list T –
  match rep
    | nil ⇒ fun x ⇒
    | None :: rep ⇒ fun x ⇒
      hd default x :: repr rep (tl x)
    | Some v :: rep ⇒ fun x ⇒
      v :: repr rep (tl x)
  end.
```

Implementation avoids getting stuck on variables.

Properties

Hold computationally!

$$\text{repr } l \,(\text{repr } r \; ls) = \text{repr } r \,(\text{repr } l \; ls)$$

$$\text{repr } l \,(\text{repr } l \; ls) = \text{repr } l \; ls$$

Composition with repr

```
Definition rep_plus := [Some nat].
Definition cfp : ∀ ts',
  let ts := repr rep_plus ts' in
  expr ts → expr ts.

fun ts' ⇒ cfp (repr rep_lt ts')
: ∀ ts',
  let ts:=repr rep_plus (repr rep_lt ts')
  in expr ts → expr ts.
```

```
Definition rep_lt:=[Some nat,Some bool].
Definition cflt : ∀ ts',
  let ts := repr rep_lt ts' in
  expr ts → expr ts.

fun ts' ⇒ cflt (repr rep_plus ts')
: ∀ ts',
  let ts:=repr rep_lt (repr rep_plus ts')
  in expr ts → expr ts.
```

Composition with repr

```
Definition rep_plus := [Some nat].
Definition cfp : ∀ ts',
  let ts := repr rep_plus ts' in
  expr ts → expr ts.

fun ts' ⇒ cfp (repr rep_lt ts')
: ∀ ts',
  let ts:=repr rep_plus (repr rep_lt ts')
  in expr ts → expr ts.
```

```
Definition rep_lt:=[Some nat,Some bool].
Definition cflt : ∀ ts',
  let ts := repr rep_lt ts' in
  expr ts → expr ts.

fun ts' ⇒ cflt (repr rep_plus ts')
: ∀ ts',
  let ts:=repr rep_lt (repr rep_plus ts')
  in expr ts → expr ts.
```

Composition with repr

```
Definition rep_plus := [Some nat].
Definition cfp : ∀ ts',
  let ts := repr rep_plus ts' in
  expr ts → expr ts.

fun ts' ⇒ cfp (repr rep_lt ts')
: ∀ ts',
  let ts:=repr rep_plus (repr rep_lt ts')
  in expr ts → expr ts.
```

```
Definition rep_lt:=[Some nat,Some bool].
Definition cflt : ∀ ts',
  let ts := repr rep_lt ts' in
  expr ts → expr ts.

fun ts' ⇒ cflt (repr rep_plus ts')
: ∀ ts',
  let ts:=repr rep_lt (repr rep_plus ts')
  in expr ts → expr ts.
```

$$\text{repr } l \ (\text{repr } r \ ls) = \text{repr } r \ (\text{repr } l \ ls)$$

Composition with `repr`

```
Definition rep_plus := [Some nat].
Definition cfp : ∀ ts',
  let ts := repr rep_plus ts' in
  expr ts → expr ts.

fun ts' ⇒ cfp (repr rep_lt ts')
: ∀ ts',
  let ts:=repr rep_plus (repr rep_lt ts')
  in expr ts → expr ts.
```

```
Definition rep_lt:=[Some nat,Some bool].
Definition cflt : ∀ ts',
  let ts := repr rep_lt ts' in
  expr ts → expr ts.

fun ts' ⇒ cflt (repr rep_plus ts')
: ∀ ts',
  let ts:=repr rep_lt (repr rep_plus ts')
  in expr ts → expr ts.
```

$\text{repr}\ l\ (\text{repr}\ r\ ls) = \text{repr}\ r\ (\text{repr}\ l\ ls)$

```
fun ts ⇒ @compose (expr (repr (rep_combine rep_plus rep_lt)  ts))
                  (cfp (repr rep_lt ts)) (cflt (repr rep_plus ts))
: ∀ ts, repr (repr (rep_combine rep_plus rep_lt) ts)
```

Composition with `repr`

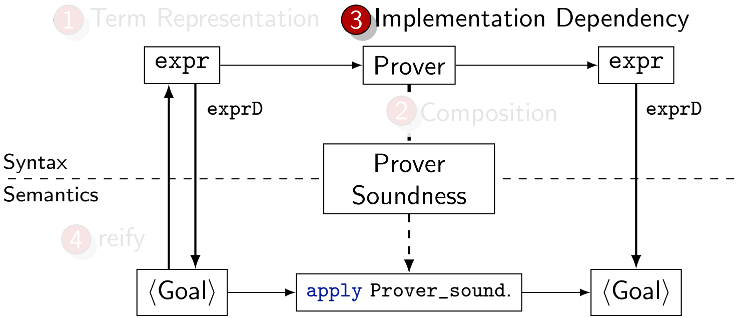
```
Definition rep_plus := [Some nat].
Definition cfp : ∀ ts',
  let ts := repr rep_plus ts' in
  expr ts → expr ts.

fun ts' ⇒ cfp (repr rep_lt ts')
: ∀ ts',
  let ts:=repr rep_plus (repr rep_lt ts')
  in expr ts → expr ts.
```

```
Definition rep_lt:=[Some nat,Some bool].
Definition cflt : ∀ ts',
  let ts := repr rep_lt ts' in
  expr ts → expr ts.

fun ts' ⇒ cflt (repr rep_plus ts')
: ∀ ts',
  let ts:=repr rep_lt (repr rep_plus ts')
  in expr ts → expr ts.
```

```
fun ts ⇒ @compose (expr (repr (rep_combine α β) ts))
                  (cfp (repr β ts)) (cflt (repr α ts))
: ∀ ts, repr (repr (rep_combine α β) ts)
```

Well-typed if α and β are **computationally** compatible.

Outline: Implementation Dependency



The Burden of (Building) Proofs

- Dependent types are convenient...

$\forall x\, y, \{x = y\} + \{x \neq y\}$

```
========
if eq_nat_dec x y
then x = y else x ≠ y
> destruct (eq_nat_dec x y).
```

The Burden of (Building) Proofs

- Dependent types are convenient...

```
==========
if eq_nat_dec x y
then x = y else x ≠ y
> destruct (eq_nat_dec x y).
```

$\forall x\, y, \{x = y\} + \{x \neq y\}$

$x = y$

The Burden of (Building) Proofs

- Dependent types are convenient...

```
========================
if eq_nat_dec x y
then x = y else x ≠ y
> destruct (eq_nat_dec x y).
```

...but constructing & eliminating proofs can be expensive.

Dependent

```
Definition dep x y :=
  if eq_nat_dec x y
  then true else false.
Eval compute in (dep 10000 10000).
```

	Dep
compute	0.252
lazy	0.424
hnf	∞

The Burden of (Building) Proofs

- Dependent types are convenient...

```
============================
if eq_nat_dec x y
then x = y else x ≠ y
> destruct (eq_nat_dec x y).
```

...but constructing & eliminating proofs can be expensive

Dependent

```
Definition dep x y :=
  if eq_nat_dec x y
  then true else false.
Eval compute in (dep 10000 10000).
```

Non-dependent

nat → nat → bool

```
Eval compute in (beq_nat 10000 10000).
```

	Dep
compute	0.252
lazy	0.424
hnf	∞

The Burden of (Building) Proofs

- Dependent types are convenient...

```
==========================
if eq_nat_dec x y
then x = y else x ≠ y
> destruct (eq_nat_dec x y).
```

...but constructing & eliminating proofs can be expensive.

Dependent

```
Definition dep x y :=
  if eq_nat_dec x y
  then true else false.
Eval compute in (dep 10000 10000).
```

Non-dependent

```
Eval compute in (beq_nat 10000 10000).
```

	Dep	**Non-dep**	**Speedup**
compute	0.252	0.012	**∼21x**
lazy	0.424	0.044	**∼10x**
hnf	∞	41	∞

The Burden of (Building) Proofs

- Dependent types are convenient...

```
========================
if eq_nat_dec x y
then x = y else x ≠ y
> destruct (eq_nat_dec x y).
```

...but constructing & eliminating proofs can be expensive.

Dependent

```
Definition dep x y :=
  if eq_nat_dec x y
  then true else false.
Eval compute in (dep 10000 10000).
```

Non-dependent

```
Eval compute in (beq_nat 10000 10000).
```

	Dep	Non-dep	Speedup
compute	0.252	0.012	$\sim$**21x**
lazy	0.424	0.044	$\sim$**10x**
hnf	∞	41	

15% overall speedup!

The Burden of (Building) Proofs

- Dependent types are convenient...

```
========================
if beq_nat x y
then x = y else x ≠ y
> destruct (beq_nat x y).
```

...but constructing & eliminating proofs can be expensive.

Dependent

```
Definition dep x y :=
  if eq_nat_dec x y
  then true else false.
Eval compute in (dep 10000 10000).
```

Non-dependent

```
Eval compute in (beq_nat 10000 10000).
```

	Dep	**Non-dep**	**Speedup**
compute	0.252	0.012	∼**21x**
lazy	0.424	0.044	∼**10x**
hnf	∞	41	∞

The Burden of (Building) Proofs

- Dependent types are convenient...

Annoying to reason about!

```
============================
if beq_nat x y
then x = y else x ≠ y
> destruct (beq_nat x y).
```

nothing...

...but constructing & eliminating proofs can be expensive.

Dependent

```
Definition dep x y :=
  if eq_nat_dec x y
  then true else false.
Eval compute in (dep 10000 10000).
```

Non-dependent

```
Eval compute in (beq_nat 10000 10000).
```

	Dep	Non-dep	Speedup
compute	0.252	0.012	**~21x**
lazy	0.424	0.044	**~10x**
hnf	∞	41	∞

Getting Simple Proofs

Non-Dependent Functions

```
========================
match beq_nat x y with
  | true ⇒ x = y
  | false ⇒ x ≠ y
end
> case_eq (beq_nat x y).
  { rewrite beq_nat_true_iff. ... }
  { ...  }
```

Getting Simple Proofs

- **Idea** Connect functions and their specs with type classes!

Non-Dependent Functions

```
========================
match beq_nat x y with
  | true ⇒ x = y
  | false ⇒ x ≠ y
end
> case_eq (beq_nat x y).
  { rewrite beq_nat_true_iff. ... }
  { ...  }
```

Simple Proofs with Type Classes

- **Idea** Connect functions and their specs with type classes!

Non-Dependent Functions

```
========================
match beq_nat x y with
  | true ⇒ x = y
  | false ⇒ x ≠ y
end
> case_eq (beq_nat x y).
  { rewrite beq_nat_true_iff. ... }
  { ...  }
```

Type Class

Like *ssreflect*

```
Inductive reflect (P Q : Prop)
  : bool → Type :=
| refl_true : P → reflect P Q true
| refl_false : Q → reflect P Q false.

Class Reflect (exp : bool) (P Q : Prop)
:= { _Reflect : reflect P Q exp }.

Instance Reflect_beq_nat : ∀ x y,
  Reflect (beq_nat x y) (x = y) (x ≠ y).
Proof. ...  Qed.

Ltac consider e :=
  let c := constr:(_ : Reflect f _ _) in
  case c.
```

Simple Proofs with Type Classes

- **Idea** Connect functions and their specs with type classes!

Non-Dependent Functions

```
========================
match beq_nat x y with
  | true ⇒ x = y
  | false ⇒ x ≠ y
end
> case_eq (beq_nat x y).
  { rewrite beq_nat_true_iff. ... }
  { ...  }
```

Type Class

```
Inductive reflect (P Q : Prop)
  : bool → Type :=
| refl_true : P → reflect P Q true
| refl_false : Q → reflect P Q false.

Class Reflect (exp : bool) (P Q :  Prop)
:= { _Reflect : reflect P Q exp }.

Instance Reflect_beq_nat : ∀ x y,
  Reflect (beq_nat x y) (x = y) (x ≠ y).
Proof. ...  Qed.

Ltac consider e :=
  let c := constr:(_ : Reflect f _ _) in
  case c.
```

Simple Proofs with Type Classes

- **Idea** Connect functions and their specs with type classes!

Non-Dependent Functions

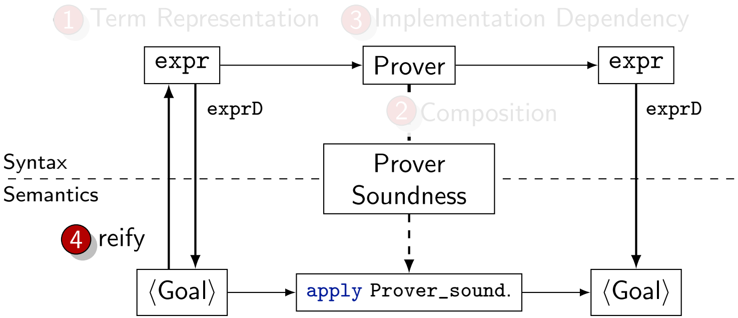
```
========================
match beq_nat x y with
  | true ⇒ x = y
  | false ⇒ x ≠ y
end
> consider (beq_nat x y); assumption.
> Qed.
```

Type Class

```
Inductive reflect (P Q : Prop)
  : bool → Type :=
| refl_true : P → reflect P Q true
| refl_false : Q → reflect P Q false.

Class Reflect (exp : bool) (P Q : Prop)
:= { _Reflect : reflect P Q exp }.

Instance Reflect_beq_nat : ∀ x y,
  Reflect (beq_nat x y) (x = y) (x ≠ y).
Proof. ... Qed.

Ltac consider e :=
  let c := constr:(_ : Reflect f _ _) in
  case c.
```

Type class resolution finds the spec!

Outline: Reification



Reifying Coq Terms

- Reify pure & separation logic expressions **with binders**.

$$\text{reify} : \langle\text{coq-term}\rangle \rightarrow (\text{ts} : \text{types} \times \text{funcs} \times \text{expr ts})$$

Reifying Coq Terms (Ltac)

reify : $\langle$coq-term$\rangle$ $\to$ (ts : types $\times$ funcs $\times$ expr ts)

Ltac Reification

✓ Single language (Ltac)

```
Ltac reify e ts fs us k :=
  match e with
  | ?X ⇒ is_evar X ;
     let t := type of X in
     let T := reifyType types t in
     get_var ts us T X ltac:(fun us v ⇒
       k us fs (@UVar types v))
  | @eq ?T ?e1 ?e2 ⇒
    let T := reifyType types T in
    reify e1 ts fs us ltac:(fun us fs e1 ⇒
    reify e2 ts fs us ltac:(fun us fs e2 ⇒
    k us fs (@Equal ts T e1 e2)))
  | fun x ⇒ @eq ?T (@?e1 x) (@?e2 x) ⇒
    let T := reifyType types T in
    reify e1 ts fs us ltac:(fun us fs e1 ⇒
    reify e2 ts fs us ltac:(fun us fs e2 ⇒
    k us fs (@Equal ts T e1 e2)))
  | ...
```

Reifying Coq Terms (Ltac)

$$\text{reify} : \langle\text{coq-term}\rangle \rightarrow (\text{ts} : \text{types} \times \text{funcs} \times \text{expr ts})$$

Ltac Reification

- ✓ Single language (Ltac)
- ✗ CPS and `idtac` for debugging
- ✗ 2nd order matching for binders

Continuation

```
Ltac reify e ts fs us k :=
  match e with
  | ?X ⇒ is_evar X ;
     let t := type of X in
     let T := reifyType types t in
     get_var ts us T X ltac:(fun us v ⇒
       k us fs (@UVar types v))
  | @eq ?T ?e1 ?e2 ⇒
    let T := reifyType types T in
    reify e1 ts fs us ltac:(fun us
    reify e2 ts fs us ltac:(fun us
    k us fs (@Equal ts T e1 e2)))
  | fun x ⇒ @eq ?T (@?e1 x) (@?e2 x) ⇒
    let T := reifyType types T in
    reify e1 ts fs us ltac:(fun us fs e1 ⇒
    reify e2 ts fs us ltac:(fun us fs e2 ⇒
    k us fs (@Equal ts T e1 e2)))
  | ...
```

Second-order match

Reifying Coq Terms (Ltac)

reify : ⟨coq-term⟩ → (ts : types × funcs × expr ts)

Ltac Reification

- ✓ Single language (Ltac)
- ✗ CPS and `idtac` for debugging
- ✗ 2nd order matching for binders
- ✗ **Slow!**
  - Re-type-check same term

```
Ltac reify e ts fs us k :=
  match e with
  | ?X ⇒ is_evar X ;
     let t := type of X in
     let T := reifyType types t in
     get_var ts us T X ltac:(fun us v ⇒
       k us fs (@UVar types v))
  | @eq ?T ?e1 ?e2 ⇒
    let T := reifyType types T in
    reify e1 ts fs us ltac:(fun us fs e1 ⇒
    reify e2 ts fs us ltac:(fun us fs e2 ⇒
    k us fs (@Equal ts T e1 e2)))
  | fun x ⇒ @eq ?T (@?e1 x) (@?e2 x) ⇒
    let T := reifyType types T in
    reify e1 ts fs us ltac:(fun us fs e1 ⇒
    reify e2 ts fs us ltac:(fun us fs e2 ⇒
    k us fs (@Equal ts T e1 e2)))
  | ...
```

Reifying Coq Terms (Ltac)

reify : ⟨coq-term⟩ → (ts : types × funcs × expr ts)

Ltac Reification

- ✓ Single language (L
- ✗ CPS and `idtac` for debugging
- ✗ 2nd order matching for binders
- ✗ Slow!
  - Re-type-check same term

```
constr:(@Func ts f ls)
```

```
Ltac reify e ts fs us k :=
  match e with
  | ?X ⇒ is_evar X ;
    let t := type of X in
                   pe types t in
                   ltac:(fun us v ⇒
                   types v))

    let T := reifyType types T in
    reify e1 ts fs us ltac:(fun us fs e1 ⇒
    reify e2 ts fs us ltac:(fun us fs e2 ⇒
    k us fs (@Equal ts T e1 e2)))
  | fun x ⇒ @eq ?T (@?e1 x) (@?e2 x) ⇒
    let T := reifyType types T in
    reify e1 ts fs us ltac:(fun us fs e1 ⇒
    reify e2 ts fs us ltac:(fun us fs e2 ⇒
    k us fs (@Equal ts T e1 e2)))
  | ...
```

Reifying Coq Terms (Ltac)

reify : ⟨coq-term⟩ → (ts : types × funcs × expr ts)

Ltac Reification

- ✓ Single language (L
- ✗ CPS and `idtac` for debugging
- ✗ 2nd order matching for binders
- ✗ Slow!
  - Re-type-check same term

```
Ltac reify e ts fs us k :=
  match e with
  | ?X ⇒ is_evar X ;
    let t := type of X in
                  pe types t in
                  ( ltac:(fun us v ⇒
                   types v))

                                      e1 ⇒
                                      e2 ⇒
    k us fs (@Equal ts T e1 e2)))
  | fun x ⇒ @eq ?T (@?e1 x) (@?e2 x) ⇒
    let T := reifyType types T in
    reify e1 ts fs us ltac:(fun us fs e1 ⇒
    reify e2 ts fs us ltac:(fun us fs e2 ⇒
    k us fs (@Equal ts T e1 e2)))
  | ...
```

`constr:(@Func ts f ls)`

Type check multiple times

Reifying Coq Terms (Plugin)

reify : $\langle$coq-term$\rangle \rightarrow$ (ts : types $\times$ funcs $\times$ expr ts)

Ltac Reification

- ✓ Single language (Ltac)
- ✗ CPS and `idtac` for debugging
- ✗ 2nd order matching for binders
- ✗ Slow!
  - Re-type-check same term

No second order matching

Plugin-based Reification

- ✓ OCaml + debugging
- ✓ Manipulate open terms

Reifying Coq Terms (Plugin)

reify : $\langle$coq-term$\rangle \rightarrow$ (ts : types $\times$ funcs $\times$ expr ts)

Ltac Reification

- ✓ Single language (Ltac)
- ✗ CPS and `idtac` for debugging
- ✗ 2nd order matching for binders
- ✗ Slow!
  - Re-type-check same term

Plugin-based Reification

- ✓ OCaml + debugging
- ✓ Manipulate open terms
- ✗ Can't return values to Ltac
- ✗ Bad functor integration

Reifying Coq Terms (Plugin)

reify : ⟨coq-term⟩ → (ts : types × funcs × expr ts)

Ltac Reification

- ✓ Single language (Ltac)
- ✗ CPS and `idtac` for debugging
- ✗ 2nd order matching for binders
- ✗ Slow!
  - Re-type-check same term

Plugin-based Reification

- ✓ OCaml + debugging
- ✓ Manipulate open terms
- ✗ Can't return values to Ltac
- ✗ Bad functor integration
- ✓ **Fast!**
  - Manipulate untyped terms

Single type check

Reifying Coq Terms (Plugin)

reify : $\langle$coq-term$\rangle \rightarrow$ (ts : types $\times$ funcs $\times$ expr ts)

Ltac Reification

- ✓ Single language (Ltac)
- ✗ CPS and `idtac` for debugging
- ✗ 2nd order matching for binders
- ✗ Slow!
  - Re-type-check

Plugin-based Reification

- ✓ OCaml + debugging
- ✓ Manipulate open terms
- ✗ Can't return values to Ltac
- ✗ Bad functor integration
- ✓ Fast!
  - Manipulate untyped terms

100x speedup

40% overall

Conclusions

Automated Verification Framework in Coq

BEDROCK

http://plv.csail.mit.edu/bedrock/